Report No. 319

IMPLICIT ENUMERATION ALGORITHM OF INTEGER PROGRAMMING ON ILLIAC IV

by

Toshihide Ibaraki
Saburo Muroga

January 20, 1969

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

IMPLICIT ENUMERATION ALGORITHM OF INTEGER PROGRAMMING ON ILLIAC IV

by

Toshihide Ibaraki

and

Saburo Muroga

January 20, 1969

This work was supported in part by the Department of Computer Science, University of Illinois, Urbana, Illinois, and in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center, under Contract No. US AF 30(602)4144.

[illegible]

[illegible]

Implicit enumeration algorithm of integer programming on ILLIAC IV

1. Introduction

In this paper, let us consider the following integer programming problem;

minimize
$$\sum_{j=1}^{n} c_j x_j \tag{1}$$

subject to
$$a_{io} + \sum_{j=1}^{n} a_{ij} x_j \geq 0 \qquad i = 1, 2, \ldots, m, \tag{2}$$

where $c_j \geq 0$ for $j = 1, 2, \ldots, n$, and where each variable x_j assumes only non-negative integer values. Furthermore, the value of each x_j may be assumed to be limited to 1 or 0 only, without loss of generality because any non-zero integer may be expressed as

$$x_k 2^k + x_{k-1} 2^{k-1} + \ldots + x_1 2^1 + x_0$$

with variables $x_k, x_{k-1}, \ldots, x_0$ which assume the value, 1 or 0.

This programming problem has been attracting intense attention from people in the various fields, because a number of problems which can not be formulated as linear programming problem can be formulated as integer programming problems. Those problems include knapsack problems, traveling salesman problems, covering problems, job-shop-scheduling problems, personnel-assignment problems, fixed-cost problems and so forth.[2][4] Logical design problems of optimum switching networks also have been formulated as integer programming problems.[12][13][3]

The integer programming problems appear far more difficult than the linear programming problems, judging from the available reports[5][7][10][3] on the computational experience, though a considerable number of algorithms have been proposed to date. These algorithms except Gomory's are completely different from the simplex method for linear programming.

Probably the most well known and tested algorithms are Gomory's cutting plane method[9] and the implicit enumeration approach[1][8][6][10].

In this paper, we will consider the implementation of the implicit enumeration algorithm on ILLIAC IV computer, to evaluate the effectiveness of parallel computation available with ILLIAC IV. Although only the case of a dense matrix (i.e. a matrix with many non-zero elements) is considered in this report, the speed-up expected from the parallel computation appears remarkable, and encourages further investigation in this respect.

The implicit enumeration algorithm have a few different versions. T. Ibaraki, T. K. Liu, C. R. Baugh and S. Muroga[10] improved these versions (except Geoffrion's latest work[7]). The program code based on their version was named ILLIP[11] (ILLinois Integer Programming). The implicit enumeration considered here is the simplified version of that of [10]. Therefore all the proofs and further detailed discussion will be found in or derived from the argument in [10].

Typically it is estimated that the entire computation is speeded up by about 100 times by using the parallel computation scheme provided by ILLIAC IV, compared with the serial computation available with an ordinary commercial computer. In the above factor, the speed-up due to the improvement of switching time of logic gates is not taken into consideratic since the exact figure about the hardware speed-up does not seem available yet.

2. Outline of Implicit Enumeration

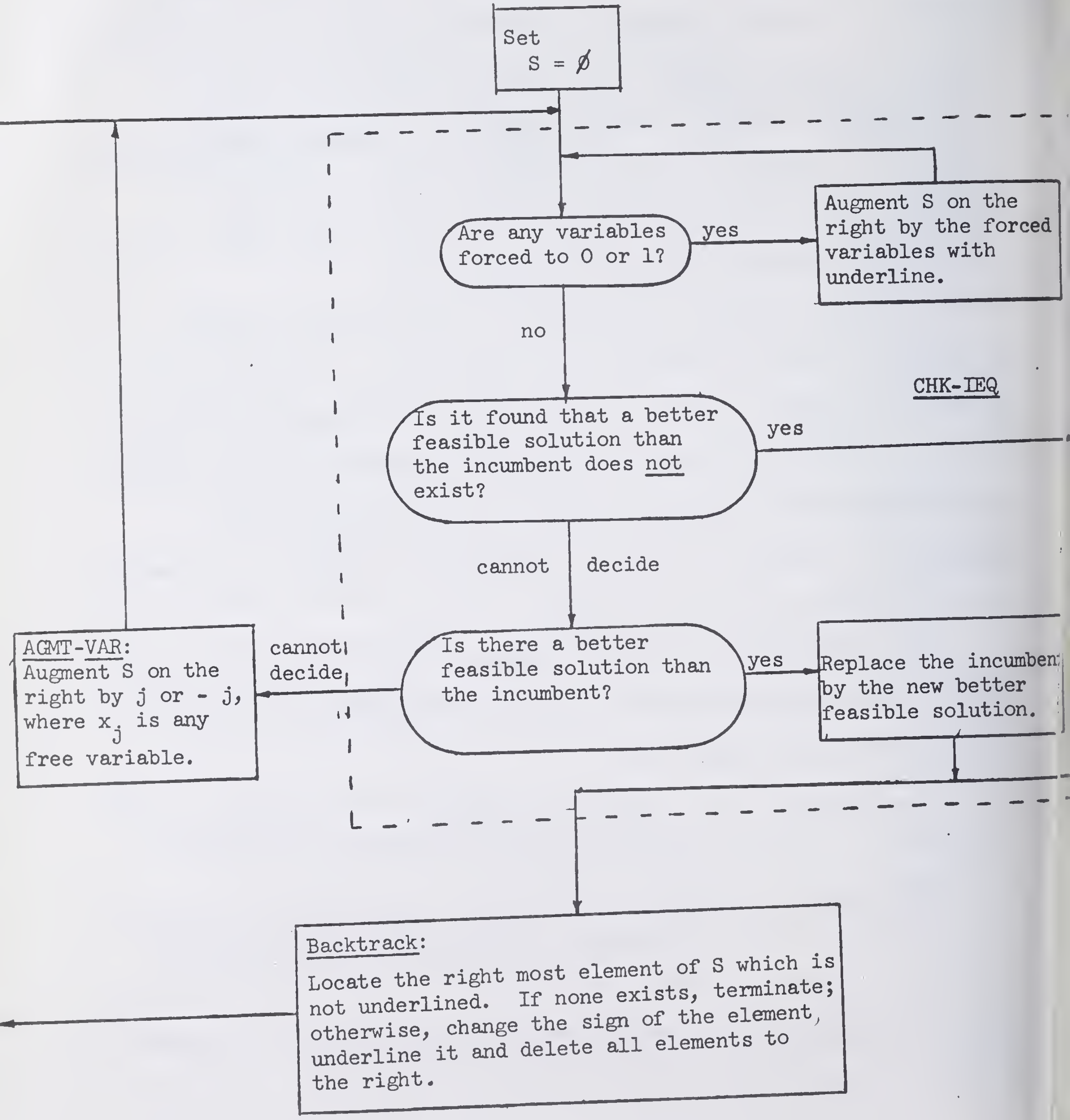
Let us begin with several definitions. Any $\vec{x}$, whose coordinates are 0 or 1, is called a solution. A solution that satisfies constrants(2) is called a feasible solution, and a feasible solution that minimizes $\vec{c} \cdot \vec{x}$ is called an optimal (feasible) solution. A partial solution S is defined as an assignment of the values, one and zero, to a subset of n variables. Any variable not assigned a value in S is called free. The set of indices of free variables is denoted by F(S).

We adopt the notational convention that the symbol j denotes $x_j = 1$ and the symbol -j denotes $x_j = 0$. Hence, if $n = 5$ and $S = \{3,5, -2\}$, $x_3 =1$, $x_5=1$ $x_2=0$, and x_1 and x_4 are free. The order in which the elements of S are written will be used to represent the order in which the variables corresponding to the elements are set to 1 or 0 by the algorithm. A completion of a partial solution S is defined as a solution that is derived from S by specifing all free variables.

The whole scheme of implicit enumeration is illustrated in Fig. 1. Given a current partial solution S, subroutine CHK-IEQ detects some of the following three S-conditions:

S-condition (a)	S forces free variables to be specified in order to obtain feasible completions of the present partial solution S,
S-condition (b)	there is no feasible completion of S or even if there are some, they are not better (smaller $\vec{c} \cdot \vec{x}$ than the incumbant) feasible completion of S, or
S-condition (c)	the best feasible completion of S is found and it is better than the incumbent.

Fig. 1 Implicit Enumeration Algorithm


Set S = ∅
Are any variables forced to 0 or 1?
yes
Augment S on the right by the forced variables with underline.
no
CHK-IEQ
Is it found that a better feasible solution than the incumbent does not exist?
yes
cannot decide
Is there a better feasible solution than the incumbent?
yes
Replace the incumben by the new better feasible solution.
cannot decide
AGMT-VAR: Augment S on the right by j or - j, where x_j is any free variable.
Backtrack: Locate the right most element of S which is not underlined. If none exists, terminate; otherwise, change the sign of the element, underline it and delete all elements to the right.

If S forces some variables to be specified (S-condition (a)), we augment S by the variable specified to that value with underline. In case of S-condition (b), we simply backtrack. However, if S-condition (c) occurs, we replace the incumbent with this feasible completion before backtracking. Finally, if none of S-conditions are detected, we go to AGMT-VAR to augment the partial solution by a certain variable which is selected according to the rule which will be explained later, without underline.

In our calculation, the objective function value is taken care of by an extra inequality:

$$\bar{z} - 1 - \vec{c}\,\vec{x} \geq 0 \,, \tag{3}$$ [*]

where $\bar{z}$ is the objective function value for the incumbent. The value of $\bar{z}$ is updated whenever the incumbent is replaced. Then if we consider inequality (3) together with (2), any feasible solution is better than the incumbent, thereby eliminating in the program checking of whether the obtained feasible solution is better than the incumbent or not.

The detection of the S-conditions is done by making use of the following three columns.

$$y_i(S) = \sum_{j \notin F(S)} a_{ij} x_j$$

$$u_i(S) = y_i(S) + \sum_{\substack{j \in F(S) \\ a_{ij} > 0}} a_{ij}$$

$$\ell_i(S) = y_i(S) + \sum_{\substack{j \in F(S) \\ a_{ij} < 0}} a_{ij}$$

$$i = 0, 1, 2, \ldots, m,$$

---

* Assumes that we want to obtain a single optimum solution for a given integer programming problem.

where i = 0 stands for the inequality defined by (3). Note that $y_i(S)$ is the lefthand side value of the i th inequality for the present value of the partial solution and $u_i(S)$ and $\ell_i(S)$ are the maximum and the minimum values of $y_i(S)$ over all completions of S, respectively. Then follow the next properties which is used for the detection of the S-conditions in CHK-IEQ.

(1) If $u_i(S) < 0$ for some i, then the i-th inequality cannot be satisfied for any completion of S. This implies that the present partial solution S is infeasible.

(2) If $|a_{ij}| > u_j(S)$ and x_j is free, then

$x_j = 0$ must hold if $a_{ij} < 0$

and $x_j = 1$ must hold if $a_{ij} > 0$.

(3) If $\ell_i(S) \geq 0$ for some i, the i-th inequality can never assume a negative value for any completion of S. Therefore we can eliminate the i-th inequality from further consideration as far as the completion of S is concerned, thus reducing the matrix size we are working on.

(4) If $\vec{y}(S) \geq 0$, then the completion obtained from S by setting all free variables to 0 is feasible and is the best among all the feasible completions of S.

For detailed discussion, see the reference [10].

The backtracking procedure shown in Fig. 1 is the same as that proposed by Glover [8]. An outline of the procedure may be understood from the description in Fig. 1.

Subroutine AGMT-VAR in our program works as follows. This is a simplified version of the method discussed in [10]. Let us prepare a row called $\vec{p}$ in which p_j, for each $j \in F$, represents the number of inequalities which satisfies $\ell_i(S') \geq 0$, where S' is S with $x_j = 1$ augmented. Among all p_j, $j \in F$, let p_{j_0} assume the maximum value. Then AGMT-VAR augments the current partial solution S with $x_{j_0} = 1$, resulting in a new partial solution, for which the check of the S-conditions by CHK-IEQ will be subsequently applied.

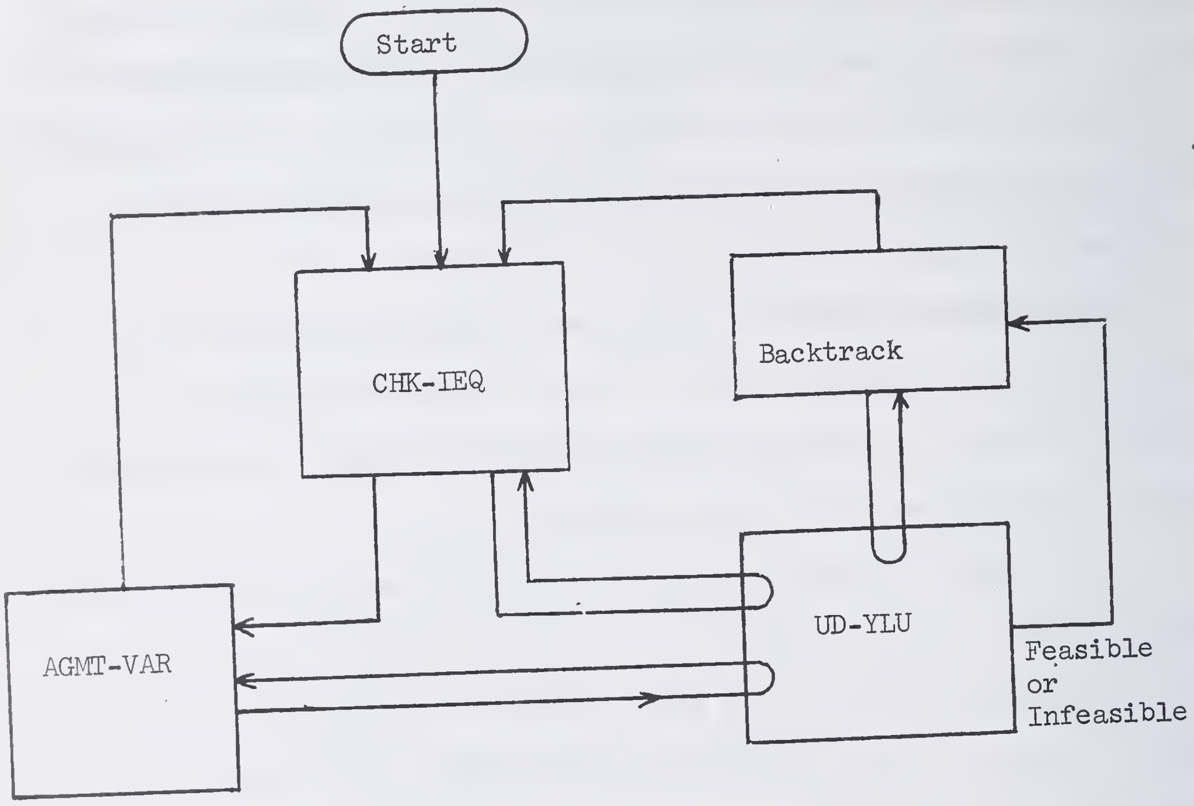
Any scheme of AGMT-VAR will let the algorithm converge in a finite number of iterations. Here the above scheme is selected because it seems reasonably efficient in spite of its simplicity, as anticipated from the computational experience in [10].

In addition to these subroutines, CHK-IEQ, backtrack and AGMT-VAR, another subroutine called UD-YLU* is included, as shown in Fig. 2. This UD-YLU updates the value of $\vec{y}$, $\vec{\ell}$ and $\vec{u}$ whenever the partial solution is modified. Therefore this subroutine is called from each of the above subroutines, CHK-IEQ, backtrack and AGMT-VAR, Upon entering UD-YLU, the new values of $\vec{y}$, $\vec{\ell}$, $\vec{u}$ are calculated, and the feasibility ($\vec{y} \geq 0$) or the infeasibility ($u_i < 0$ for some i) of the partial solution is detected in this subroutine. Usually after the execution of UD-YLU the program control returns to the subroutine from which UD-YLU was called but in the feasible or infeasible case, the execution of backtrack follows (after replacing the incumbent if feasible.).

The whole scheme is illustrated in Fig. 2. More detailed description of each individual subroutine will be given in the next section, using flow charts for the illustration of the implementation on ILLIAC IV.

* UD-YLU is scattered in other subroutines in Fig. 1. But for the sake of ease in programming, we make it an independent subroutine.

Fig. 2. Linkage of subroutines



3. Implicit Enumeration on ILLIAC IV

In this section, first the overall storage allocation is discussed. Then using this allocation scheme, each subroutine is described with a flow chart. Actual number of each instruction used in the procedure is counted for each of both parallel (Illiac IV) and serial computation (an ordinary computer) cases. An estimation of speed difference between these two cases will be summarized in the next section.

For the sake of simplicity, consider a problem with 1000 variables and 256 inequalities. The density of non-zero coefficients of the given problem is high and hence no packing scheme of data is employed.

Fig. 3 illustrates the storage allocation scheme in ILLIAC IV. Each inequality is assigned to 1 to 1004 of each PEM (i.e. Processing Element Memory) and data processing of each inequality is assigned to each PE (i.e. Processing Element.) As illustrated, locations 1, 2, 3 of PE serve as $\vec{u}$, $\vec{\ell}$, $\vec{y}$ respectively, whereas locations 4 ~ 1004 are used for accomodating the coefficients of problem including those of the objective function. Besides such entries as $\vec{u}$, $\vec{\ell}$, $\vec{y}$ and coefficients of the given problem, four kinds of the arrays $\vec{p}$, $\vec{x}$, $\vec{s}$, $\vec{opt}$ (the last three of these will be defined in the following.) are also provided. These arrays are allocated in the locations of 1005 through 1020. Each of these four has 1000 entries so that each entry may correspond to a variable of the given problem. $\vec{x}$ keeps the value of the current partial solution in such a way that the value of x_j is stored in the j-th entry of $\vec{x}$. The entries of $\vec{x}$ are counted from the first entry of the first row to the last entry of the fourth row of $\vec{x}$, as shown in Fig. 3. (The shaded areas in Fig. 3 are empty.) $\vec{s}$ also represents the current partial solution in a

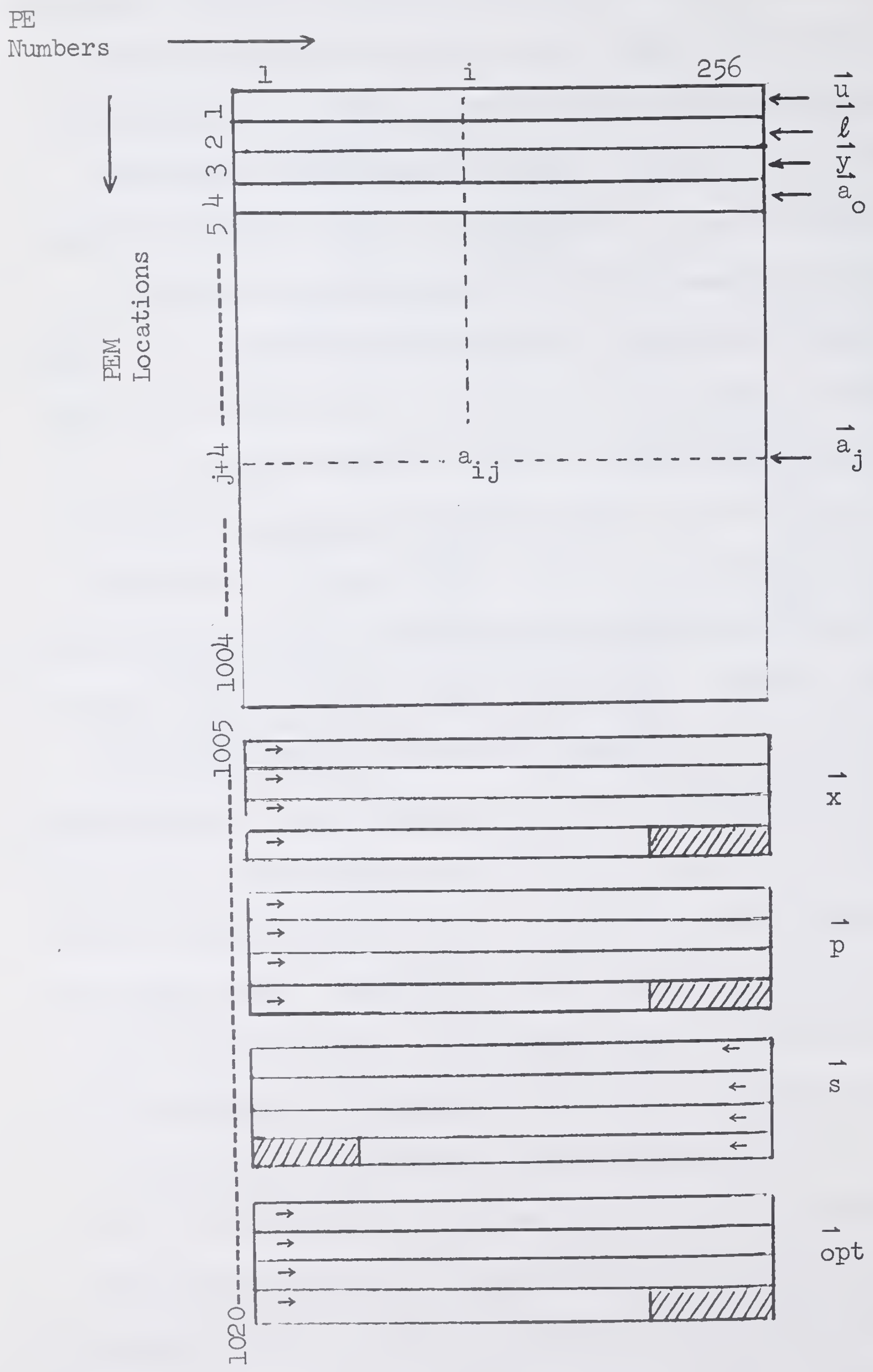


Fig. 3. Storage Allocation

different way from $\vec{x}$. Each assignment of variable is entered from the beginning of $\vec{s}$ according to the order of their generation in the course of the computation. Note that the packing of $\vec{s}$ start from the right hand side and from the uppermost row, as shown in Fig. 3. In other words, the array $\vec{s}$ is used to keep track of the enumeration status and to operate the backtracking procedure on it. $\vec{p}$ is used to store the parameters calculated in AGMT-VAR as explained in Section 2. $\vec{opt}$ contains the incumbent solution. The information of the value of x_j is also stored in the j-th location of $\vec{p}$ and $\vec{opt}$, in the same way as $\vec{x}$.

With these preparations, the detailed description of each subroutine is given next. Note, however, that this report is a preliminary study and more efficient program might be conceivable by utilizing the capability of ILLIAC IV more carefully.

3-1 CHK-IEQ

Fig. 4 is a flow chart of subroutine CHK-IEQ. Roughly speaking, it checks whether there exists a_{ij} such that $|a_{ij}| > u_i$ for every free variable x_j. If the condition is satisfied by some free variable, the variables are set to 0 or 1 according to the sign of a_{ij}, followed by the call of UD-YLU to update $\vec{y}$, $\vec{\ell}$ and $\vec{u}$.

The speedup by the parallel computation is mainly due to the portion where the condition $|a_{ij}| > u_i$ and the sign of a_{ij} is checked (marked as A in Fig. 4). For example the existence of $|a_{ij}| > u_i$ and $a_{ij} < 0$ can be found by a procedure shown in Fig. 5. First the sum of two arrays $\vec{u}$ and $\vec{a}_j$ is retained in all registers of 256 PE's. Then the sign bits of these 256 registers are sent to a Mode register, by which the existence of negative entries in all the above 256 registers is checked by one instruction. This process needs 1L and 1A operations*

* In the following, L stands for Load, A for Add, S for Subtract, ST for STore and R for Routing.

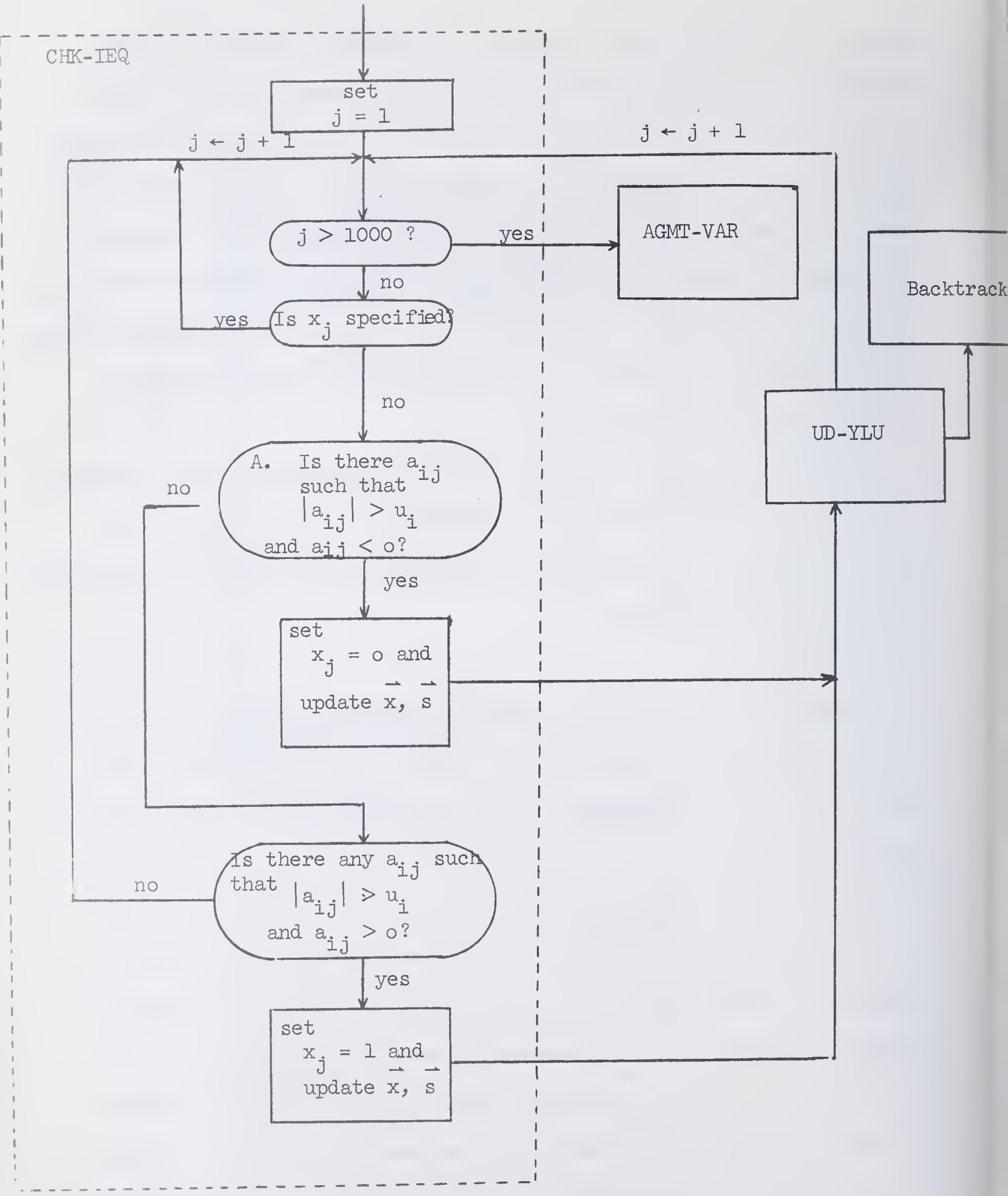
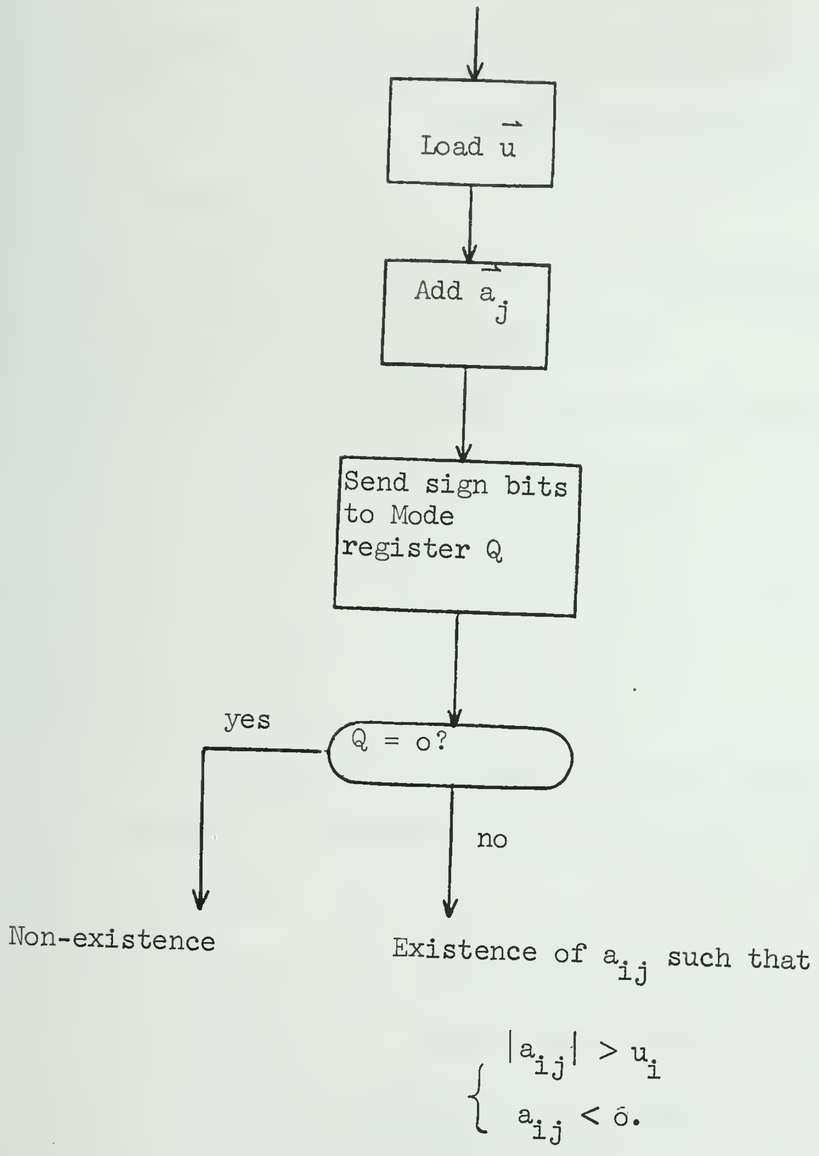

CHK-IEQ
set
j = 1
j ← j + 1
j ← j + 1
j > 1000 ?
yes
AGMT-VAR
no
Backtrack
yes
Is x_j specified?
no
UD-YLU
A. Is there a_{ij} such that $|a_{ij}| > u_i$ and $a_{ij} < o$?
no
yes
set $x_j = o$ and update $\vec{x}, \vec{s}$
Is there any a_{ij} such that $|a_{ij}| \geqslant u_i$ and $a_{ij} > o$?
no
yes
set $x_j = 1$ and update $\vec{x}, \vec{s}$

Fig. 4. Flow-Chart CHK-IEQ

Fig. 5. Detailed description of portion A in Fig. 4.



Load $\vec{u}$
Add $\vec{a}_j$
Send sign bits to Mode register Q
Q = o?
yes
no
Non-existence
Existence of a_{ij} such that
$\begin{cases} |a_{ij}| > u_i \\ a_{ij} < 0. \end{cases}$

whereas the serial computation needs 256 L and A and also 256 sign checks. We count the number of instructions L, ST, A, S, R only, since these consume most of execution time.

Consequently the total number of instructions used to perform the procedure in Fig. 4 for each x_j is given by

$$(L,S),$$

if x_j is not free, and

$$(4L,\ A,\ 3S)$$

if a free variable x_j is set to neither 0 nor 1. If x_j is set to 0,

$$(4L,\ 3ST,\ 2A,\ 2S)$$

steps are necessary and if x_j is set to 1,

$$(5L,\ 3ST,\ 2A,\ 3S)$$

steps. The above numbers assume the parallel computation.

On the other hand, if the serial computation is adopted, for each free x_j,

$$(L,S)$$

$$(514L,\ 256A,\ 258S),$$

$$(259L,\ 3ST,\ 257A,\ 2S)$$

$$\text{and}\quad (515L,\ 3ST,\ 257A,\ 258S)$$

steps are needed corresponding to the four cases in which x_j is not free, x_j is specified to 0 and 1, and no assignment is performed on x_j, respectively.

3-2 AGMT-VAR

AGMT-VAR works as shown in Fig. 6. As roughly explained in Section 2, AGMT-VAR first calculates the parameter array $\vec{p}$, and then picks up the variable x_{j_0} which has the maximum value in $\vec{p}$. Then x_{j_0} is set to 1. After updating $\vec{y}$, $\vec{\ell}$, $\vec{u}$ accordingly, CHK-IEQ is entered to check the new partial solution.

The portion B indicated in Fig. 6 is shown in more detail in Fig. 7. First, assuming that a free variable x_j is set to 1, the new value of $\vec{\ell}$, i.e., $\vec{\ell}'$, is calculated. The sign of each ℓ_i' is then extracted (denoted by C in Fig. 7) and then put in another register in each PE.

This process can be done by first transfering the sign bits of registers containing the values of ℓ_i' to one of the mode register. Then the contents of the mode register is distributed back to each register in the corresponding PE.

In the next block D, the contents of 256 registers are summed up to count the number of ℓ_i' with non-negative values. This is done by using the "logarithmic sum" scheme known among ILLIAC IV people. In other words, a number in each odd-numbered PE added to a number of the PE on its right by the first instruction. And then addition of each adjacent pair of sums is repeated until the total sum is formed at the right most PE. To obtain the total of 256 PE's, (8A, 48R) instructions are necessary.

The rest of the flow chart is self-explanatory.

An estimation of the number of major instructions necessary to perform the work in portion B is as follows:

Fig. 6. AGMT-VAR

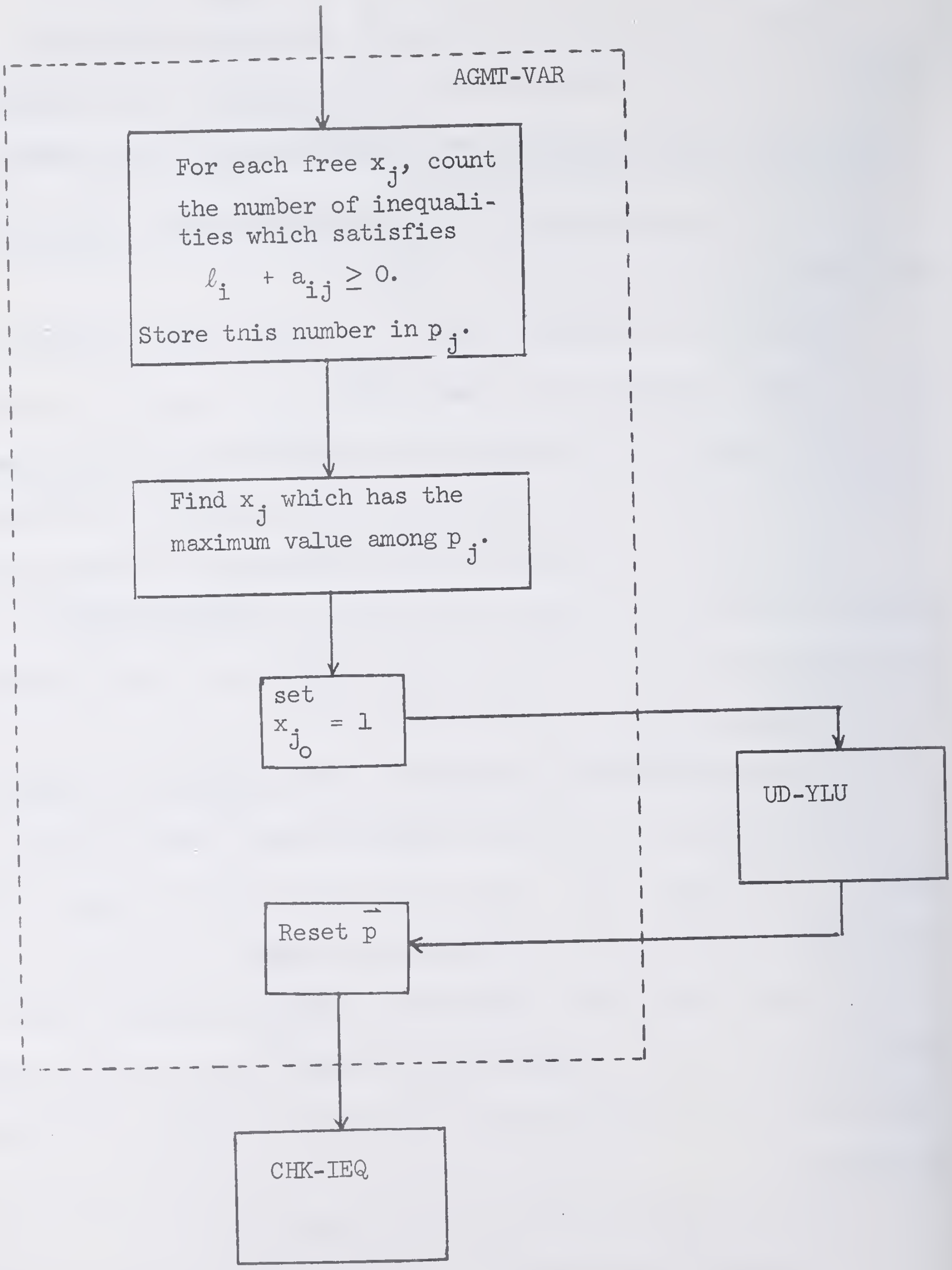
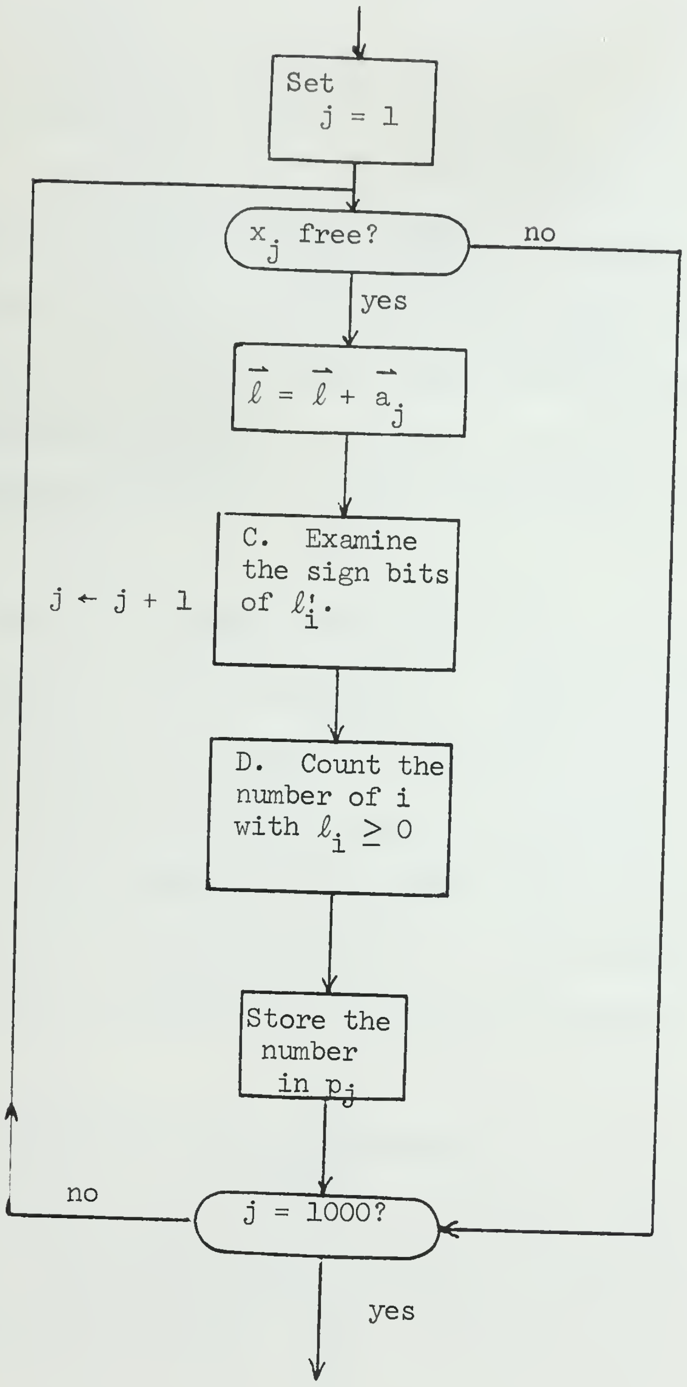

AGMT-VAR
For each free x_j, count the number of inequalities which satisfies $\ell_i + a_{ij} \geq 0$. Store this number in p_j.
Find x_j which has the maximum value among p_j.
set $x_{j_o} = 1$
UD-YLU
Reset $\vec{p}$
CHK-IEQ

Fig. 7. Portion B of AGMT-VAR



Set $j = 1$
x_j free?
no
yes
$\vec{\ell} = \vec{\ell} + \vec{a}_j$
C. Examine the sign bits of ℓ_i'.
$j \leftarrow j + 1$
D. Count the number of i with $\ell_i \geq 0$
Store the number in p_j
no
$j = 1000$?
yes

(1) When x_j is free;

(1L, 1ST, 9A, 48R) : parallel computation,

and (256L, 1ST, 512A) : serial computation,

(2) when x_j is specified;

(1L, 1S) : both parallel and serial computations.

After obtaining the values of $\vec{p}$ by applying the process to all the variables, the rest of the flow chart needs

(L, 3ST, A, 9S, 48R) : parallel computation

(257L, 2ST, A, 257S) : serial computation.

3-3 Backtrack

Subroutine Backtrack is entered from UD-YLU, which will be explained in Section 3-4, after finding the feasibility or infeasibility. Backtrack operates, on the array $\vec{s}$ which keeps the value of variables in the current partial solution in such form that each specified variable is stacked from the beginning of $\vec{s}$ in the order of variable assignment made during the computation, either with underline or without underline. If the value of a variable is positive, it is underlined, and, otherwise, not underlined.

As shown in Fig. 8, the program first locates the last negative (not underlined) entry. Then the sign of the entry is switched to positive, simultaneously deleting all the entries which follow after that variable. Of course, this is the restatement of the operation of Backtrack illustrated in Fig. 1.

Fig. 9 is a more detailed flow chart of Backtrack, which should be self-explanatory.

Fig. 8. Backtrack

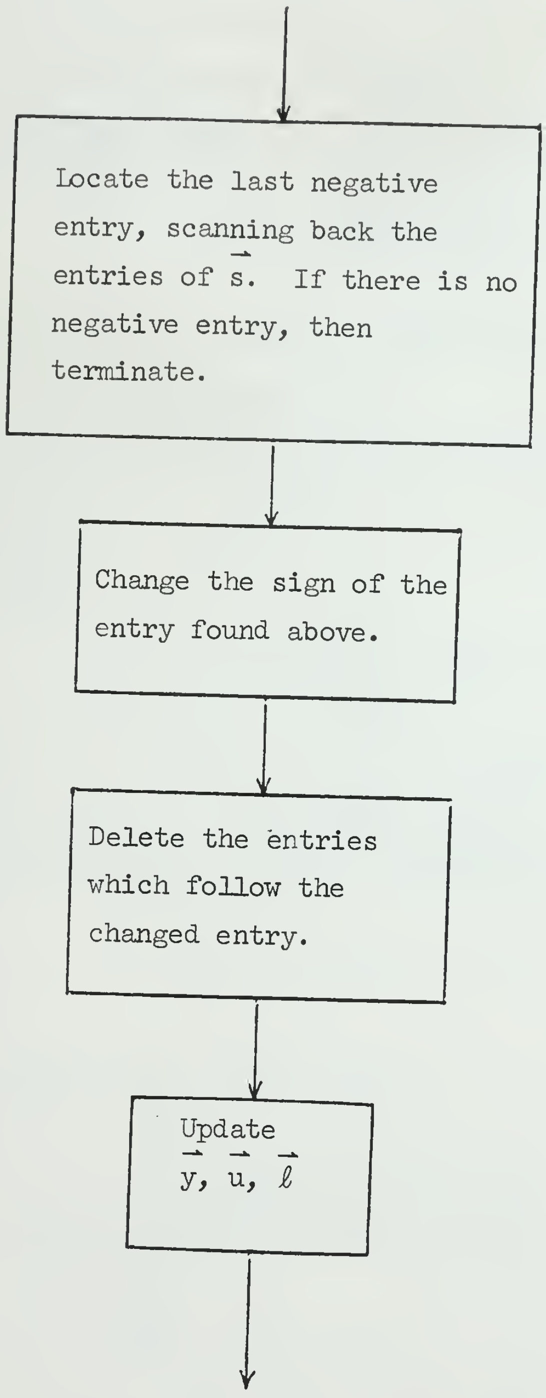
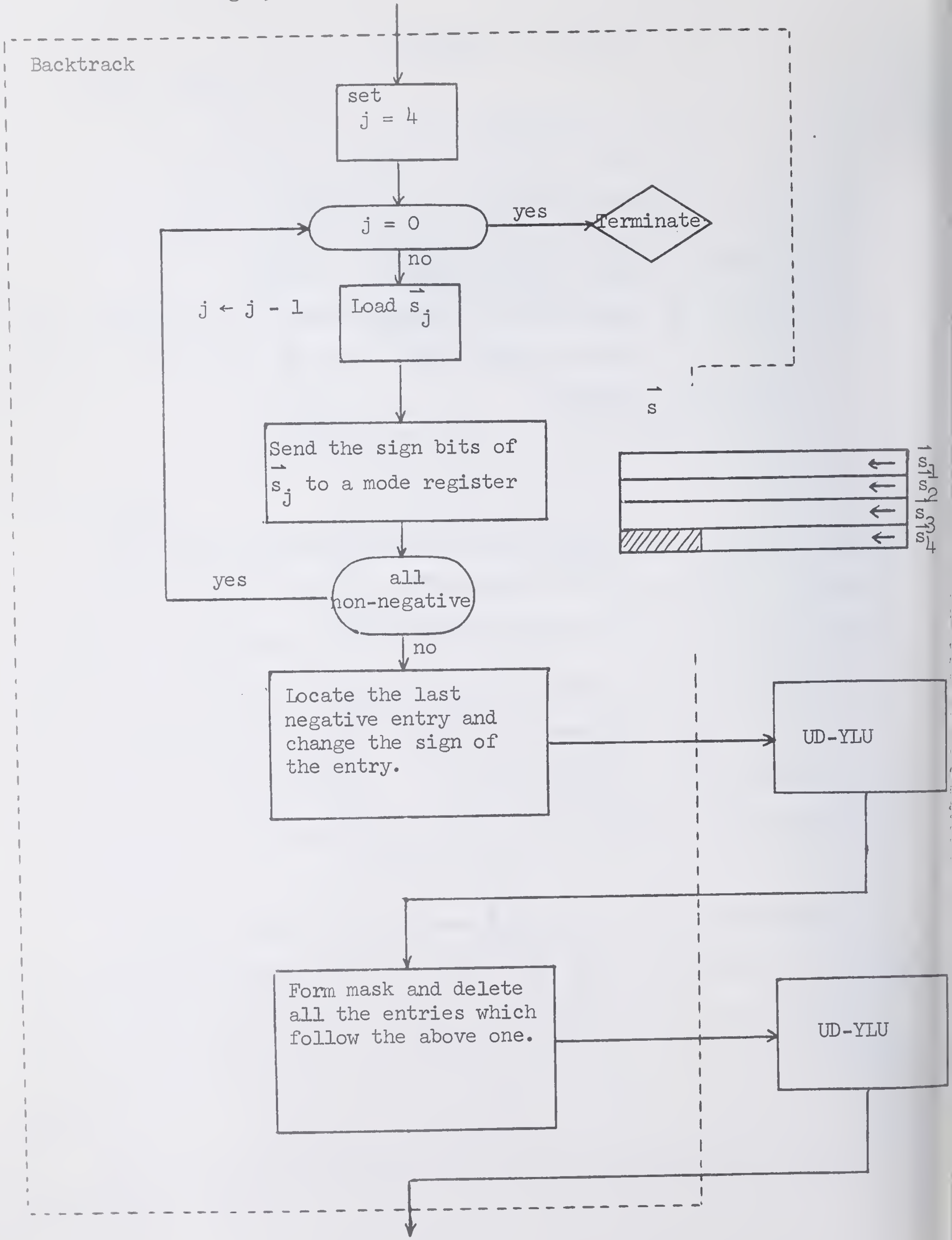

Locate the last negative entry, scanning back the entries of $\vec{s}$. If there is no negative entry, then terminate.
Change the sign of the entry found above.
Delete the entries which follow the changed entry.
Update $\vec{y}$, $\vec{u}$, $\vec{\ell}$

Fig. 9. Flow Chart of Backtrack


Backtrack
set j = 4
j = 0
yes
Terminate
no
j ← j - 1
Load $\vec{s}_j$
$\vec{s}$
Send the sign bits of $\vec{s}_j$ to a mode register
$\vec{s}_1$
$\vec{s}_2$
$\vec{s}_3$
$\vec{s}_4$
yes
all non-negative
no
Locate the last negative entry and change the sign of the entry.
UD-YLU
Form mask and delete all the entries which follow the above one.
UD-YLU

Now assume that, on the average, the loop in Fig. 9 is repeated 2.5 times to locate the last negative entry. The number of major instructions used to go through this subroutine is;

(3.5L, 2ST) : parallel computation

(641L, 2ST) : serial computation.

3-4 UD-YLU

This subroutine shown in Fig. 10 is most efficiently speeded up by making use of the parallel computation.

Depending on the situation in which this subroutine is called, there are six cases;

$$x_j: \quad 0 \to 1,\ 0 \to *,\ 1 \to 0,\ 1 \to *,\ * \to 1,\ * \to 0,$$

where * means that the variable is not specified (free). In each case, the updating procedure of $\vec{y}$, $\vec{\ell}$ and $\vec{u}$ is slightly different, but not significantly. For example, if x_j is changed from 1 to *, the new $\vec{y}'$, $\vec{\ell}'$ and $\vec{w}'$ are determined from the present values $\vec{y}$, $\vec{\ell}$ and $\vec{u}$, by

$$y'_i = y_i - a_{ij}$$

$$\ell'_i = \begin{cases} \ell_i - a_{ij}, & \text{if } a_{ij} > 0 \\ \ell_i, & \text{if } a_{ij} \leq 0 \end{cases}$$

$$u'_i = \begin{cases} u_i, & \text{if } a_{ij} \geq 0 \\ u_i - a_{ij}, & \text{if } a_{ij} < 0. \end{cases}$$

Other cases have similar changes.

The detailed procedures only for two of the above six cases are illustrated in Fig. 10.

Besides updating $\vec{y}$, $\vec{\ell}$, $\vec{u}$, (labeled as "updating" in Fig. 10) this subroutine checks the infeasibility and feasibility of the new partial solution by observing the sign of the new $\vec{u}$ and $\vec{y}$. As mentioned in Section 2, if at least one of $\vec{u}$ is negative, then the partial solution is infeasible. In addition, if $\vec{y} \geq 0$ is satisfied, then the partial solution is feasible. The exits, feasible and infeasible, indicate these two cases. If feasible and infeasible cases occur, appropriate actions are taken, though they are not described in detail since these cases occur very rarely in the entire computation. (In most of time, the program control leaves UD-YLU through the exit "updating".)

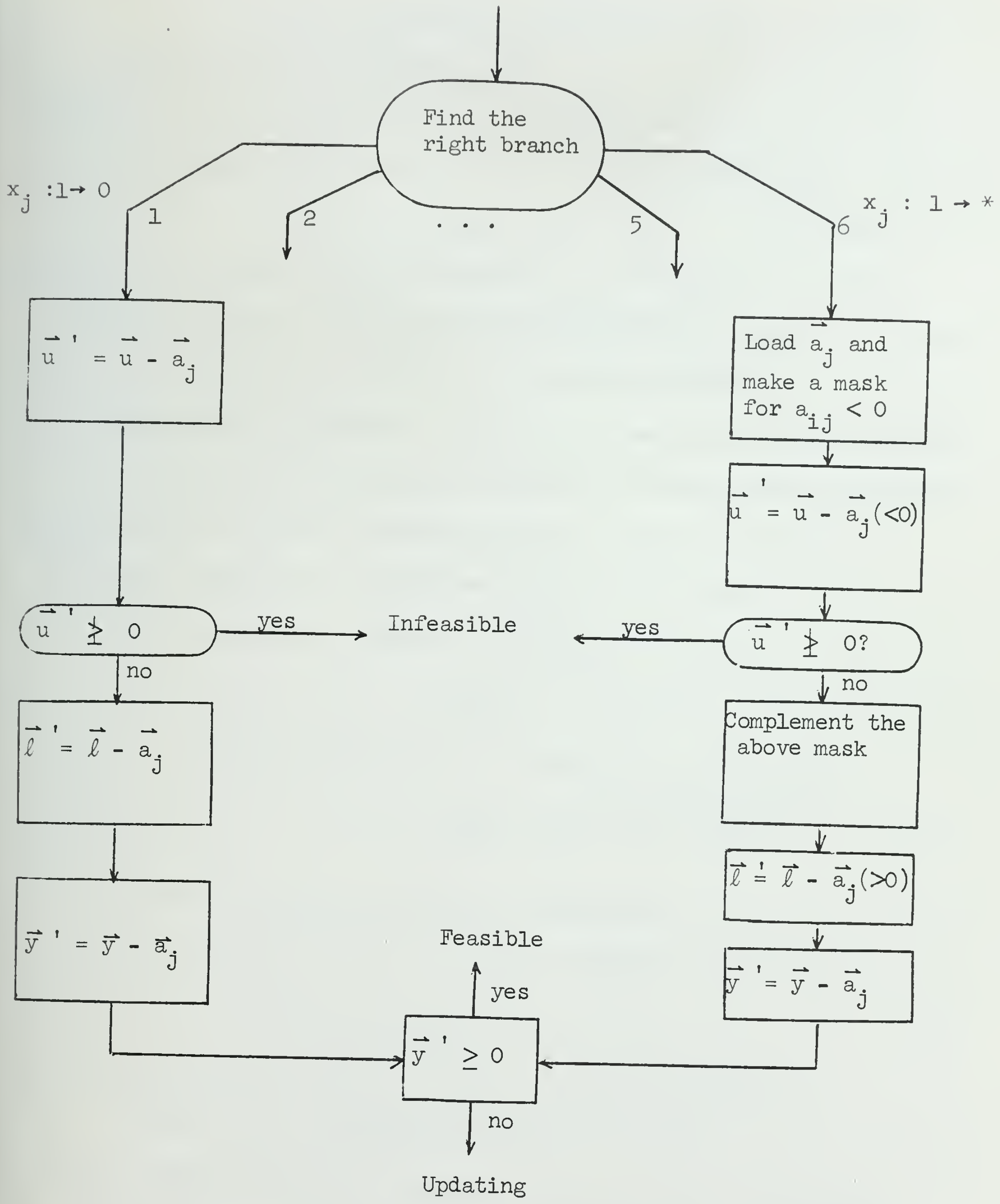
With these remarks, the flow chart of Fig. 10 should be easily understood.

The number of major instructions used in a branch is

$$\begin{aligned} &(3L,\ 3ST,\ 3S) : \text{ parallel computation} \\ 256\ &(3L,\ 3ST,\ 3S) : \text{ serial computation,} \end{aligned}$$

where those cases, feasible and infeasible, are assumed not to occur.

Fig. 10. UD-YLU


Find the right branch
$x_j : 1 \to 0$
1
2
. . .
5
6
$x_j : 1 \to *$
$\vec{u}' = \vec{u} - \vec{a}_j$
Load $\vec{a}_j$ and make a mask for $a_{ij} < 0$
$\vec{u}' = \vec{u} - \vec{a}_j(<0)$
$\vec{u}' \ngeq 0$
yes
Infeasible
yes
$\vec{u}' \ngeq 0$?
no
no
$\vec{\ell}' = \vec{\ell} - \vec{a}_j$
Complement the above mask
$\vec{\ell}' = \vec{\ell} - \vec{a}_j(>0)$
$\vec{y}' = \vec{y} - \vec{a}_j$
Feasible
yes
$\vec{y}' = \vec{y} - \vec{a}_j$
$\vec{y}' \geq 0$
no
Updating

4. Estimation of Computation Time on ILLIAC IV

Using the number of instructions estimated for each subroutine in Section 3, this section derives an overall speed-up due to the parallel computation capability provided by ILLIAC IV, over the conventional serial computation.

Before the conclusion, let us obtain the total number of instructions necessary for going through CHK-IEQ and AGMT-VAR in a typical situation derived from our computational experience[10][3]. In Section 3, the estimation of operations pertiment to each variable x_j only, not the estimation for the total work for these subroutines, were given.

Now assume that, before entering CHK-IEQ, 500 out of 1000 variables are free, and 50 among these 500 free variables are specified in CHK-IEQ, 25 variables to 0 and 25 variables to 1. Then the total number of instructions used to handle 1000 variables is

$$\begin{aligned} &500\ (L,S) + 450\ (4L,A,3S) + 25\ (4L,\ 3ST,\ 2A,\ 2S) \\ &+ 25\ (5L,3ST,2A,3S) \\ &= (2125L,\ 150ST,\ 600A,\ 1075S) \end{aligned}$$

in case of the parallel computation. On the other hand, the serial computation needs the following number of instructions under the same condition.

$$\begin{aligned} &1000\ (L,S) + 450\ (514L,3ST,257A,2S) \\ &+ 25\ (259L,3ST,257A,2S) + 25\ (515L,3ST,257A,258S) \\ &= (250650L,150ST,128050A,122600S). \end{aligned}$$

Adopting the same assumption that 500 variables are free, the total number of instructions used to go through AGMT-VAR is;

500 (L,ST,9A,48R) + 500 (L,S)
+ (L,3ST,A,9S,48R)
= (1001L,503ST,4501A,509S,24048R)

: parallel computation,

500 (256L,ST,512A) + 500 (L,S)
+ (257L,2ST,A,257S)
= (128757L,502ST,256001A,757S)

: serial computation.

With these figures, let us proceed to estimate the total speedup in the entire implicit enumeration procedure. There are two main paths in the overall flow chart shown in Fig. 1 :

(a) path A : CHK-IEQ → AGMT-VAR

(b) path B : CHK-IEQ → $\begin{matrix}\text{Feasibile}\\ \text{Infeasible}\end{matrix}$ → Backtrack.

First consider path A. Under the assumption that 50 free variables are specified, we will call UD-YLU 51 times to pass through path A. The total number of instructions used in CHK-IEQ, AGMT-VAR and 51 UD-YLU is

(3259L,806ST,5101A,1737S,24048R)

: parallel computation,

(418575L,39820ST,384051A,162525S)

: serial computation.

In order to get an estimation for path B, let us further assume, that the infeasibility of the current partial solution is detected in the middle of CHK-IEQ, i.e., after 25 free variables are specified, and then Backtrack is initiated. Also assume that the last 50 variables in the current partial solution are underlined. Then for this path B, a half of CHK-IEQ, one Backtrack and 76 UD-YLU are used. Note that we neglect the feasible case of the partial solution because this occurs very rarely in the usual computation. The total number of instructions used in path B is:

(1291L,377ST,300A,763S)

: parallel computation,

(183566L,57677ST,64025A,118900S)

: serial computation.

Taking the figures presently available for ILLIAC IV, 200 nano seconds for each of L, S, A, ST and 80 nano seconds for R, let us consider that the execution of R is 2.5 times faster than others, and the execution time for L, ST, A, S which are the same, is used as a unit time.

Then, according to the above argument, path A needs

20522 unit times : parallel

1004971 unit times : serial,

while path B needs

2731 unit times : parallel

424168 unit times : serial.

Assuming that paths A and B occur evenly as a typical example, the speed-up attained by making use of the parallel processing of ILLIAC IV is about 102 times, judging from the unit times necessary as estimated above.

5 Conclusion

This report presents a preliminary consideration in order to implement the implicit enumeration algorithm of integer programming problem on the ILLIAC IV computer, which enables us to perform the parallel processing to some extent. The tentative result shown in the report may indicate that the implementation of the algorithm on ILLIAC IV is worth while, since about 100 times speed-up suggests that a considerably large size problem, compared with the problem solvable by the serial computation, can be solved on ILLIAC IV.

Although a variety of programming gimmicks and reorganization of the algorithm may make the program more efficient, the speed-up factor is believed not to deviate too much from that obtained in this report.

It should be noticed, however, that the type of problem assumed in the discussion, i.e. the size of matrix and the dense coefficient matrix, is undoubtedly one of the most favorable types for ILLIAC IV. Therefore the future investigation of the problems of sparse matrices, in which non-zero coefficients are packed in the memory, and large size problems, (much larger than 256 inequalities and 1000 variables) in which the coefficient matrix has to be folded in the memory, complicating the indexing procedure, is essential to evaluate the parallel processing of the implicit enumeration algorithm.

Also considering that ILLIAC IV is constructed such that the unit time used in the previous section is much faster than that of conventional machines, this speedup due to hard-ware improvement should be further taken into account, to estimate realistically the value of ILLIAC IV.

It is hoped that this preliminary investigation will be followed by more detailed discussion, possibly by actual coding, to verify the usefullness of ILLIAC IV in this area.

The authors are grateful for the encouragement and support of Professor D. Slotnick.

References

(1) E. Balas, "An additive algorithm for solving linear programming with zero-one variables," Operations Research, vol. 13, no. 4, pp. 517-544, July-August, 1965.

(2) M. L. Balinski, "Integer programming: methods, uses, computation," Management Science, vol. 12, no. 3, pp. 253-313, November 1965.

(3) C. R. Baugh, T. Ibaraki, T. K. Liu and S. Muroga, "Optimum network design using NOR and NOR-AND gates by integer programming," Report no. 293, Department of Computer Science, University of Illinois, January 1969.

(4) E. M. L. Beale, "Survey of integer programming," Operational Research Quarterly, vol. 16, no. 2 pp. 219-228, 1965.

(5) B. Fleischmann, "Computational experience with the algorithm of Balas," Operations Research, vol. 14, no. 1, pp. 153-155, January-February, 1966.

(6) A. M. Geoffrion, "Integer programming by implicit enumeration and Balas' Method," SIAM Review, vol. 9, no. 2, pp. 178-190, April 1967.

(7) A. M. Geoffrion, "An improved implicit enumeration approach for integer programming," The RAND Corporation, Memorandum RM-5644-PR, June 1968.

(8) F. Glover, "A multiphase-dual algorithm for the zero-one integer programming problem," Operations Research, vol. 13, no. 6, pp. 879-919, November-December, 1965.

(9) R. E. Gomory, "An all-integer integer programming algorithm," Industrial Scheduling, edited by J. R. Muth and G. L. Thompson, Prentice-Hall, 1963.

(10) T. Ibaraki, T. K. Liu, C. R. Baugh and S. Muroga, "An implicit enumeration program for zero-one integer programming," Report no. 305, Department of Computer Science, University of Illinois, January 1969.

(11) T. K. Liu, "A code for zero-one integer linear programming by implicit enumeration," Report no. 302, Department of Computer Science, University of Illinois, December 1968.

(12) S. Muroga and T. Ibaraki, "Logical design of an optimum network by integer linear programming-Part I," Report no. 264, Department of Computer Science, University of Illinois, July, 1968.

(13) S. Muroga and T. Ibaraki, "Logical design of an optimum network by integer linear programming-Part II," Report no. 289, Department of Computer Science, University of Illinois, December, 1968.